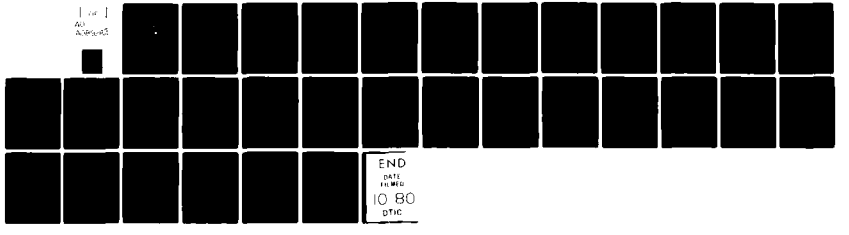
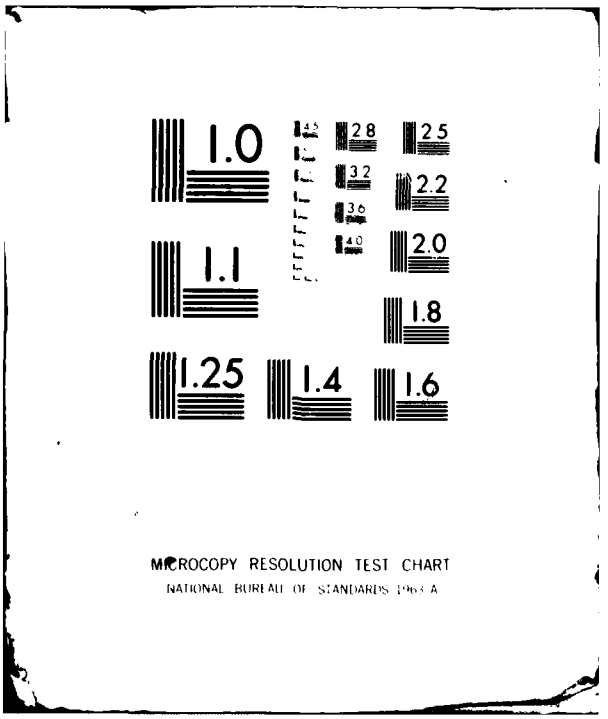
AD-A089 284 NAVAL POSTGRADUATE SCHOOL MONTEREY CA F/G 9/2
DYNAMIC LINKING IN A MICROCOMPUTER ENVIRONMENT.(U)
MAY 80 G B BLANTON, R R SCHELL
UNCLASSIFIED NPS52-80-008 NL



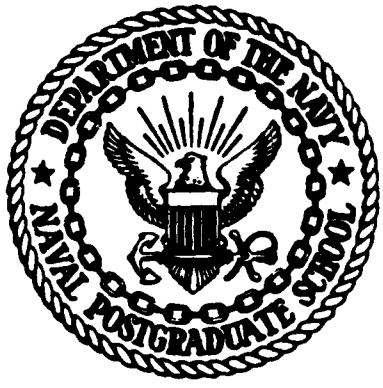
END
DATE FILMED
10 80
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

NPS-52-80-008

NAVAL POSTGRADUATE SCHOOL

Monterey, California



DYNAMIC LINKING IN A MICROCOMPUTER ENVIRONMENT

Gerald B. Blanton
Roger R. Schell

Approved for public release: distribution unlimited

Prepared for: Chief of Naval Research
Arlington, VA 22217

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral J. J. Ekelund
Superintendent

Jack Borsting
Provost

The work reported herein was supported in part by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

This report was prepared by:

Gerald B. Blanton

GERALD B. BLANTON, LT, USN

Roger R. Schell

ROGER R. SCHELL, LTCOL, USAF

Reviewed by:

GORDON H. BRADLEY, Chairman
Department of Computer Science

Released by:

William M. Tolles

WILLIAM M. TOLLES
Dean of Research

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: NPS-52-80-008	2. GOVT ACCESSION NO.: AD-A089284
	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): DYNAMIC LINKING IN A MICROCOMPUTER ENVIRONMENT	5. TYPE OF REPORT & PERIOD COVERED: Technical May 1980
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Gerald B. Blanton, Roger R. Schell	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Department of Computer Science, Naval Postgraduate School, Monterey, CA 93940	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61152N; RR000-01-10; N0001480WR00054
11. CONTROLLING OFFICE NAME AND ADDRESS: Office of Naval Research, Arlington, VA 22217	12. REPORT DATE: May 1980
	13. NUMBER OF PAGES: 28
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office): Technical Rpts	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Dynamic Linking, Linkers, Operating Systems, Micro processors, Pure Procedures

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The design of a dynamic linker for use with modern microcomputers is introduced. The basic concepts for resolving symbolic interprocedure references at execution time are reviewed, and the means to support shared data and (pure) procedures are explained. The mechanism (modules and data bases) suitable for microcomputers without the hardware features previously considered necessary for dynamic linking are described. The major implications for the associated operating system and language translators are identified.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 68 IS OBSOLETE S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

DYNAMIC LINKING IN A MICROCOMPUTER ENVIRONMENT

GERALD B. BLANTON, Lt., USN, and ROGER R. SCHELL, Lt. Col., USAF

Department of Computer Science

Naval Postgraduate School

Monterey, California

Abstract

The design of a dynamic linker for use with modern microcomputers is introduced. The basic concepts for resolving symbolic interprocedure references at execution time are reviewed, and the means to support shared data and (pure) procedures are explained. The mechanism (modules and data bases) suitable for microcomputers without the hardware features previously considered necessary for dynamic linking are described. The major implications for the associated operating system and language translators are identified.

Authors' address: Naval Postgraduate School, Code 52, Monterey, CA 93940.

Authors' telephone: (408)-646-2449

Presentation: Gerald B. Blanton will present the paper at COMPSAC '80.

Proceedings: Roger R. Schell is responsible for inclusion of the paper in the conference proceedings.

Acknowledgment: The research reported here was sponsored in part by Office of Naval Reserach Project Number NR 337-005.

I. INTRODUCTION

Dynamic linking provides a flexible and convenient way for resolving a procedure's symbolic references to external procedures and data. It is distinguished by when a symbolic reference is bound to the (virtual) address of the external object, viz., binding is deferred until the first actual reference during execution. Yet in spite of the significant benefits and more than a decade to assimilate the technology, the Multics system remains as essentially the singular working example. Several authors of articles [1] and operating systems texts [2,3] imply that a major reason for this limited use is the special hardware required. We have addressed this limitation and have developed a dynamic linker design that requires no more hardware support that is provided by modern microcomputers.

A. Background

We have used Multics [4] as the standard for the dynamic linking capabilities we would like to provide. Multics has an integrated hardware and software design with the hardware features that strongly support dynamic linking being (1) indirect addressing through memory, (2) a linkage fault during indirection, (3) segmentation, and (4) demand paging.

Although commercial microcomputers do not have the range of hardware support found in Multics, they are increasingly capable. This has naturally led to increased expectations for services, and extensive capabilities such as full-functioned languages (e.g., PL/I) and multiprogrammed, multiprocessor operating systems [5] are emerging. We believe that dynamic linking is an attractive addition to this growing set of system capabilities.

B. Objectives

We want our design to provide the usual benefits of dynamic linking. A process should include only those procedures and data actually referenced during execution, vice those included in the program. Similarly, during software development programs may be run with references to procedures (e.g., error routines) not yet coded. Even when all the referenced objects are available, they should be allocated (memory) resources only if the object is actually used, without the user predicting the actual usage before program execution.

The dynamic linker must not seriously degrade other capabilities of the system. We have taken care to accommodate (but not require) in-core sharing of (pure) procedures and data, mixed languages in a process, a flexible file system, and a security kernel structure [5] for the operating system. Programming generality is preserved in that a (dynamically linked) external reference can be used whenever an internal reference can be used.

Finally, we note that performance has been a dominant consideration. If all programs were interpreted rather than translated we clearly could simulate the Multics hardware support for dynamic linking; however, for reasons of performance we have not chosen the interpreter approach. Our major performance objective has been to reduce to an absolute minimum the processing overhead required for the second and subsequent references to an external object. Much more processing is required for the first reference that must, of course, create and save (for subsequent references) the information on the binding from symbolic reference to virtual address. The completion of this first reference is at the heart of dynamic linking.

II. BASIC CONCEPTS FOR DYNAMIC LINKING

Dynamic linking always occurs during the execution of a procedure in the address space of a process. It results from the symbolic reference to an (explicitly declared) external procedure or data object. Dynamic linking implicitly requires the notion of a segment, i.e., a logical group of information (external object) that retains distinct attributes (e.g., symbolic name) for the life of the process. Although hardware segmentation is not mandated, each segment has an identifier (unique in each process) that we will call the segment number. A segment number and an offset within the segment constitute a virtual address that can be used to reference a specific location.

On a procedure's first reference (in a process) to an external object the dynamic linker, with support from the operating system, computes the virtual address corresponding to the symbolic reference. The linker stores this binding in what we call a link. We then say the link is "snapped", and subsequent references are made through this snapped link without use of the linker. We will next examine conceptually the functions and data bases that make up the linker, and the rationale for them.

A. External Procedures

Consider what happens when a procedure segment <Caller> executes the first reference in a process corresponding to the source statement:

CALL Target

<Caller>'s object code is typically envisioned as containing:

CALL (virtual address of the entry into <Target>) (1)

at the time of execution. However, until <Target> is linked this is not

feasible, since its virtual address is not known. We might alternatively consider translating the source into

CALL <Linker> ("Target") (2)

where <Linker> is a procedure segment that calls on the underlying operating system to find the virtual address of the entry for the segment named "Target".

After determining the virtual address of <Target>, the linker could "snap the link" by overwriting statement (2) in <Caller>'s object code with statement (1), a call to the <Target>. This is essentially what is done in the traditional (static) linking loader [1]. However, to do this dynamically for a running program makes <Caller> impure, in violation of our design criteria. To address this we follow the Multics [4] example, introducing the concept of a linkage segment and a linkage pointer that points to the section (linkage table) in it for <Caller>.

Every process has a distinct (never shared) linkage table which contains an outgoing link for each external reference by <Caller>. The outgoing link for the reference to <Target> can be at a fixed offset (at translation) from the linkage pointer. Thus the object code of the pure procedure <Caller> can be of the form:

CALL (Linkage_pointer + <Target> link offset) (3)

The outgoing link is designed to invoke <Linker> on the first reference with something similar to statement (1) above. The linker then modifies (i.e., snaps) the outgoing link to invoke <Target> on subsequent references.

It might seem that the snapped link should merely be of the form of statement (2) above. However, a significant problem arises. After the transfer of control to <Target> we, of course, expect the linkage pointer to

point to the linkage table for <Target>, not <Caller>. Unfortunately, the (pure) code of <Target> cannot properly load the linkage pointer, because the appropriate virtual address is unknown at translation time. Therefore, the linker includes in the linkage table of <Target> an incoming link for the entry to <Target>. Thus the proper form for the outgoing link from <Caller> is:

CALL (<Target> incoming link) (4)

This snapped incoming link contains a statement to load the linkage pointer, followed by the form of statement (1) to invoke <Target>. This completes the invocation and dynamic linking of the external procedure <Target>.

B. External Data

An external data reference has many similarities to the above. Consider the reference in <Caller> corresponding to the source statement:

pointer := ADDRESS(Data)

The pure code in <Caller> will be very similar to statement (3) for the same reasons; in particular we expect the form:

CALL (Linkage_pointer + <Data> link offset) (5)

The unsnapped link in <Caller>'s linkage table will be similar to statement (2) for procedures. In particular the form

CALL <Linker> ("Data") (6)

will invoke the linker to obtain the virtual address of <Data> and snap the link. The snapped link is much simpler than for procedures. We need something of the form:

RETURN (virtual address of <Data>) (7)

This completes the reference to and dynamic linking of the external data segment.

III. THE MICROCOMPUTER DYNAMIC LINKER DESIGN

A. The Steps in Establishing a Link

Having examined the basic concepts of a dynamic linker, we will now investigate the features of a design for its realization. Once again the steps (Figure 1) necessary to dynamically link some external procedure <Target| entry_name> (1) to procedure <Caller> will be followed, but emphasis will be placed on design details vice linking concepts.

1. Invoking the Linker

When <Caller>'s source code is translated, any external reference to <Target> will be translated into code which transfers the execution point to an outgoing link in <Caller>'s linkage table (Caller.link). Translated code will vary from system to system; however, conceptually we desire to have a pure procedure (<Caller>'s object code) call some impure procedure (Caller.link). This will allow us at some later point in the linking process to convert the initialized outgoing link in Caller.link from code which invokes the linker to code which will result in the invocation of <Target| entry_name>.

We have mentioned that the outgoing link is initialized to invoke the linker. One method to do this would be for the outgoing link to produce a

---

(1) Entry_name represents a label within <Target> which can be referenced by an external procedure. An entry point is an offset within <Target> associated with some entry name.

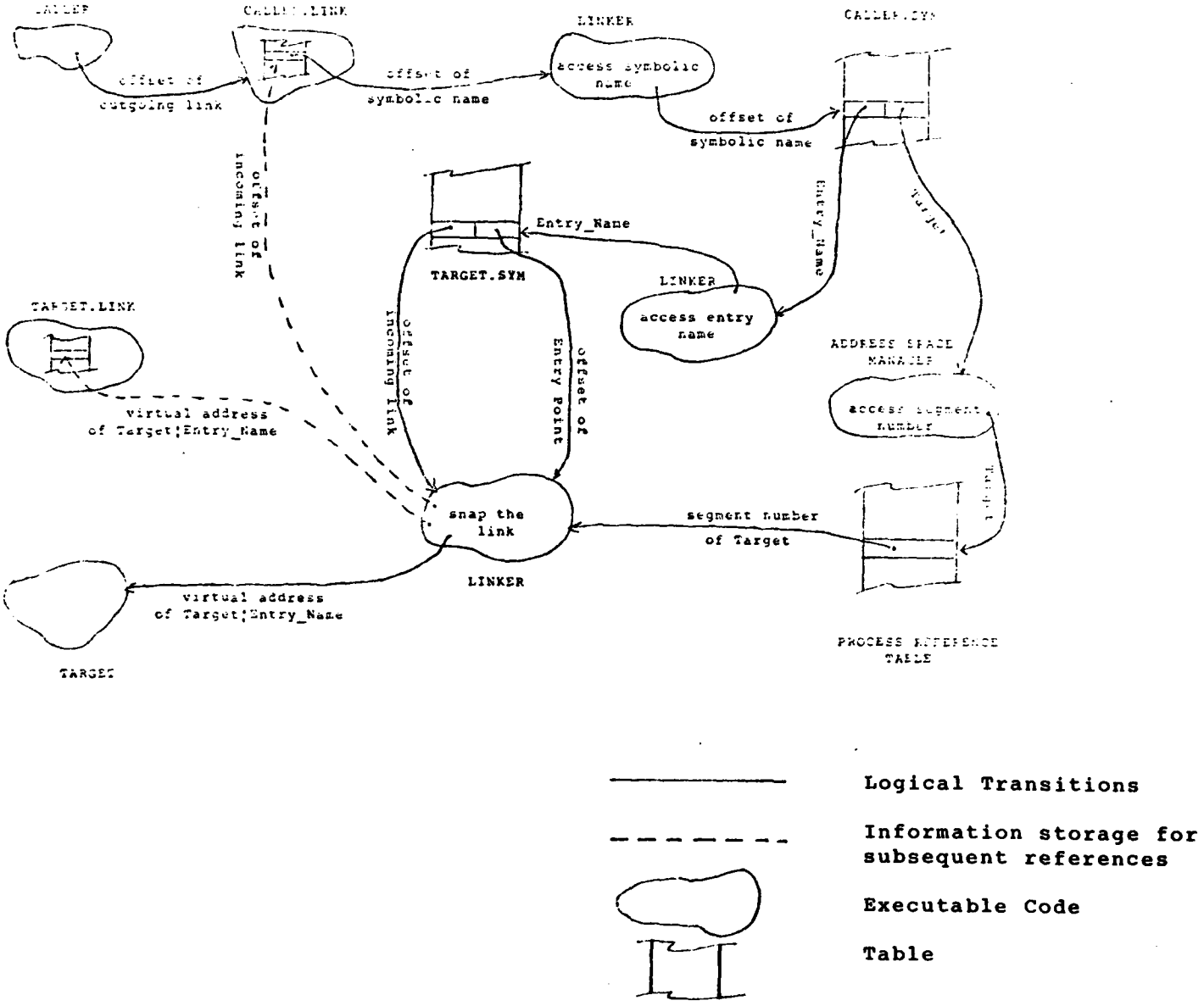


Fig. 1 - Sequence of events for snapping the link to the procedure Target|Entry_Name.

hardware fault which will invoke the linker as a fault handler. (This represents the most general of the methods available and is the method used in Multics.) However, barring this capability, the linker may be invoked via a direct jump in the outgoing link. In both these methods the linker would be passed the offset of the symbolic name <Target| entry_name> in <Caller>'s symbolic name table (Caller.sym). (An object's symbolic name table stores the symbolic names of all external references in a procedure. Additionally it will be used to retain entry names and their associated entry points utilized within the procedure.) A third mechanism to invoke the linker is via an explicit call in <Caller>'s source code passing to the linker the symbolic name of the object to be linked as a actual parameter. (This represents a special case which will be discussed later.)

2. Snapping the Link

The question naturally arises of how the linker knows where Caller.sym is located. We have provided a mechanism to resolve this problem by placing the virtual address of an object's symbolic name table in a header in the object's linkage table. (We can always find an object's linkage table since we propose to store a pointer to it in a linkage address table.) Thus the linker can access the <Target| entry_name> by utilizing the virtual address of the symbolic name table (from the linkage table header) and the offset it was passed as a formal parameter.

The linker will then pass the symbolic name <Target> to a module in the supervisor called the address space manager (which will be discussed in more detail later). At this point we will consider the address space manager a routine which, when passed a symbolic name, enters the object associated with that symbolic name in the address space of the executing process and returns

to the linker the segment number assigned to that object.

Now that we know the segment number of <Target>, we can generate a complete virtual address for <Target| entry_name> by accessing the entry point into <Target> associated with 'entry_name'. Recall that the symbolic name table of an object contains not only the symbolic names of external references, but also entry names (and their associated entry points) into the object. Thus if we can access Target.sym we can find the entry point associated with 'entry_name'. When the segment number of <Target> is returned to the linker, the linker will construct a linkage table for <Target> (if one does not already exist) to allow <Target> to engage in dynamic linking. Since the header of Target.link contains the virtual address of Target.sym, we can find the entry point corresponding to 'entry_name'.

We now have a virtual address (of the form <segment number| entry_point>) and could invoke <Target| entry_name> at this point. However, we would like to retain this virtual address to simplify subsequent invocations of <Target| entry_name>. We can accomplish this goal by storing the virtual address of <Target| entry_name> in Target.link. An incoming link has been set aside for this purpose and we can transfer control to this incoming link on subsequent calls by replacing the jump to the linker found in the outgoing link (in Caller.link) with a jump to the incoming link (for <Target| entry_name>). Thus subsequent calls will follow the sequence shown in Figure 2.

Recall that the linkage pointer is used to indicate the start of the currently executing procedure's linkage table. Thus when we start executing in <Target>, we want the linkage pointer to point to Target.link. The incoming link for <Target| entry_name> will be used to set the linkage pointer (prior to transferring control to <Target| entry_name>). This implies that

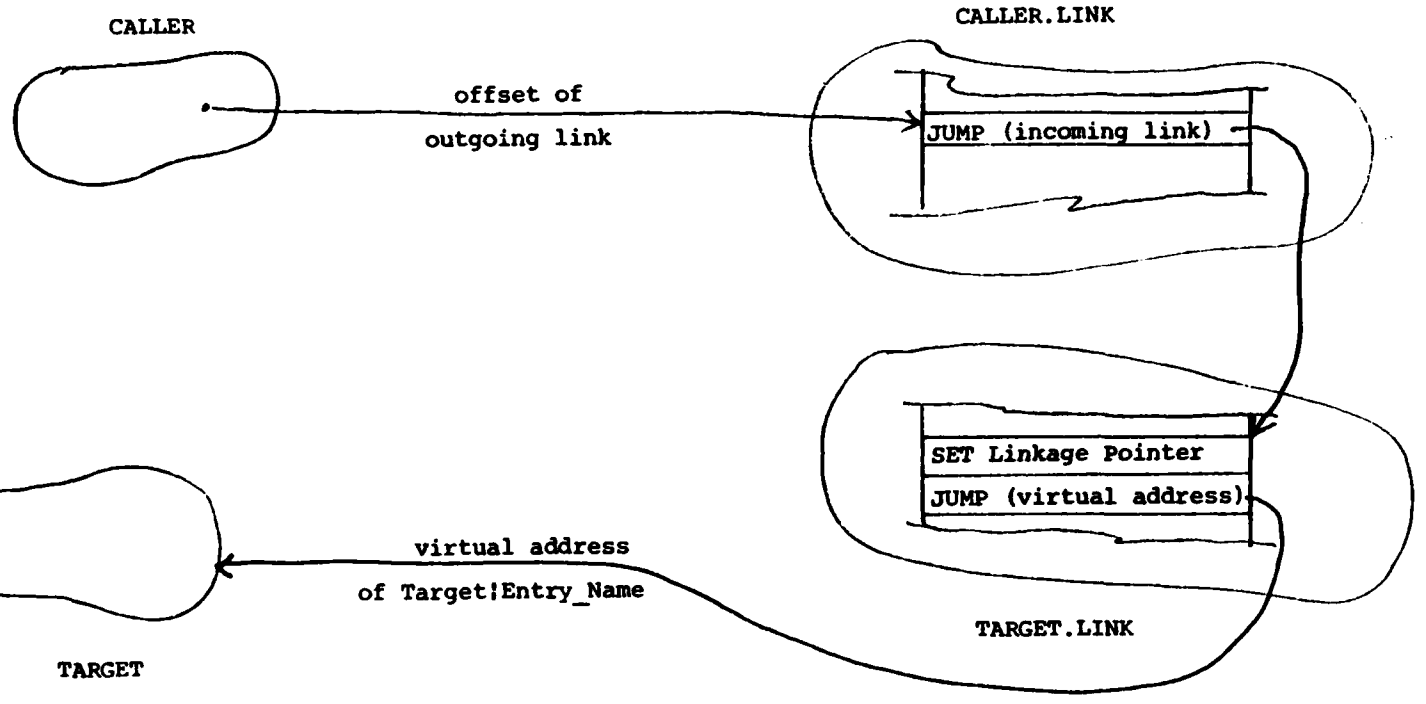


Fig. 2 - Sequence of events for subsequent references to Target|Entry_Name.

when <Caller> calls an outgoing link the linkage pointer (which points to Caller.link) must be saved and furthermore must be restored when the process execution point returns from <Target> to <Caller>. This may be done automatically by the hardware CALL/RETURN sequence or explicitly by the object code in <Caller>.

3. Linking Data

The steps to link data differ slightly from those for linking procedures. Fundamentally, data is not executed and does not, therefore, have the capability to initiate dynamic linking. Thus if <Caller> were to reference some data object, <Data| entry_name>, instead of invoking <Data| entry-name> all we really want to know is its virtual address. We therefore propose that when snapped, the outgoing link for <Data| entry_name> will load some pointer register with the virtual address of <Data| entry_name> and then return to <Caller>. The sequence for subsequent references to <Data| entry name> is shown in Figure 3.

We note that the use of entry names within data implies that the data must undergo a translation and have a symbolic name table and linkage table. These two items may be merged into one structure since the data linkage table consist of a header which (at this point) only contains a pointer to the data symbolic name table.

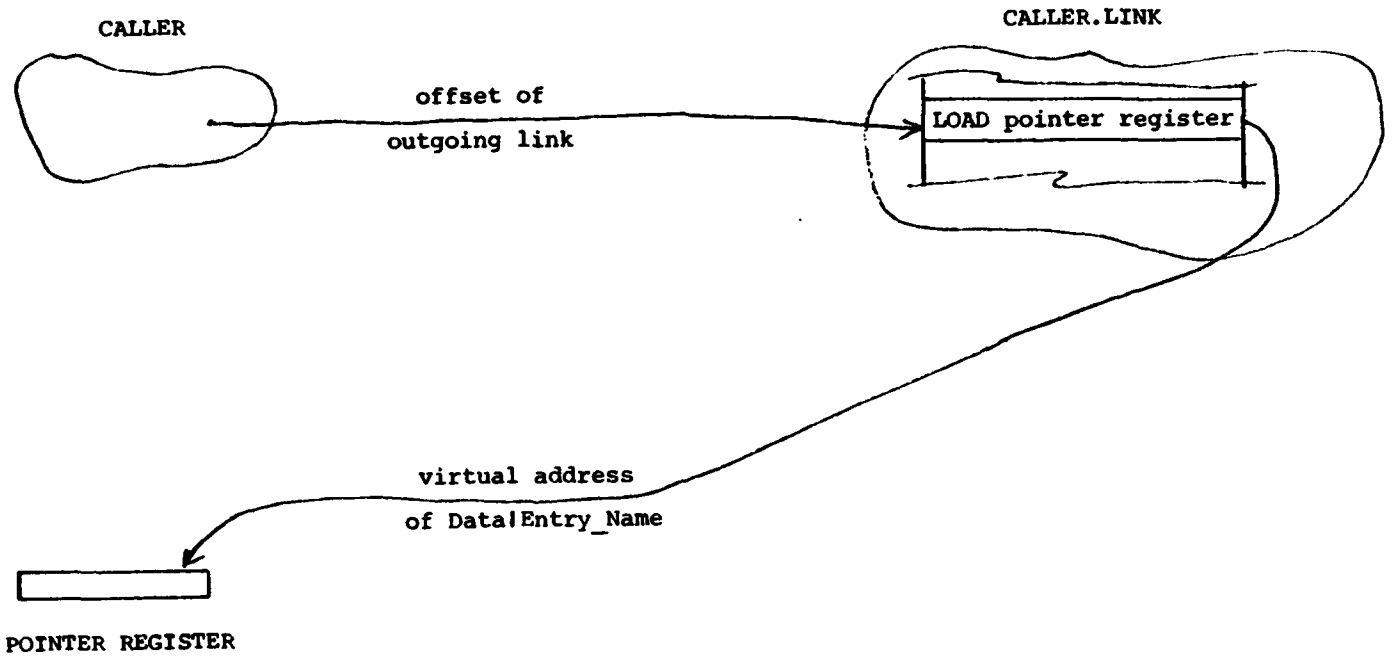


Fig. 3 - Sequence of events for subsequent references to Data!Entry_Name.

9. Unlinking

In a microcomputer environment, there may exist a limited number of segments available for use by a process (2). To prohibit this from limiting the size of programs, we may choose to unlink objects from an address space in order to allow the dynamic linking of new objects to the process.

1. Fundamentals of Unlinking

Conceptually unlinking is a trivial process but may include pitfalls which must be taken into account. To unlink, an object is selected for removal, the object's entry in the linkage address table is erased, and all outgoing links to the object are reset to their presnapped format. This implies two major points. First, any data required to reset an outgoing link must either be accessible by some means or stored in the outgoing link itself. Secondly, we must be able to find each snapped, outgoing link. It is proposed that a linked list of outgoing links referring to an object be formed with a pointer to the start of this linked list stored in the object's linkage table header. Thus, in the trivial case, unlinking an object consist of erasing the object's entry in the linkage address table, resetting each outgoing link in the object's linked list, and finally deleting the object and its linkage table from the process address space.

2. Resolving Addressing in the Combined Linkage Table

Recall that we said there were pitfalls involved in unlinking. These arise from how an object's linkage table is entered in a process address

---

(2) As an example, the Z8000 microprocessor [7] with one Memory Management Unit allows a maximum of 64 segments, some of which must be assigned to the operating system.

space. Observe that if we are concerned with unlinking, there are not adequate segments available to assign each linkage table a unique segment number. It is proposed that in such an environment, individual linkage tables be combined into a single segment which we will call the combined linkage table.

Even though the combined linkage table conserves segments, it creates addressing problems which must be taken into account. If, when removing a deleted object's linkage table from the combined linkage table, a compaction of the combined linkage table is performed, some remaining linkage tables may be relocated.

This relocation requires the following three addressing problems be resolved: (1) a relocated linkage table must have its entry in the linkage address table updated, (2) outgoing links which transfer control to an incoming link in a relocated linkage table must be updated, (3) nodes of an (unlinker) linked list pointing to outgoing links in a relocated linkage table must be updated.

A fourth addressing problem which must be resolved involves the removed object's linkage table. Each outgoing link in this linkage table is a node in a linked list and must be deleted from that linked list prior to removing the linkage table from the process address space. (We note that a doubly linked or circular linked list of outgoing links is helpful in resolving these last two addressing problems.)

3. Selecting an Object for Removal

A brief comment is in order concerning the selection of an object for removal. A procedure cannot be unlinked and removed from the address space if it has a current activation record since this would break the integrity of the return sequence from a series of one or more procedure calls. Therefore a

mechanism must be developed to test for this condition prior to selecting an object for removal. This implies, for example, that the initial program cannot be unlinked since it is always either executing or in the return sequence.

IV. SYSTEM SUPPORT

We should now like to briefly discuss operating system support for dynamic linking. It is felt that the capabilities discussed either already exist or are implementable in most microcomputers.

A. The Address Space Manager

The address space manager serves as an interface between the dynamic linker and the operating system. It will call on File System Management and (in some cases) Memory Management to obtain data which allows it to convert the symbolic name of an object into an addressable entity. Having done this, the address space manager will save this data in a table (which we will call the process reference table) to prevent unnecessary invocations of operating system routines for subsequent request to make an object addressable by a process. We note that the process reference table is a 'per process' data structure and each object entered in it has a unique set of attributes (such as segment number, access rights, etc.) with respect to that process.

When passed a symbolic name, the address space manager must be able to access the object associated with that symbolic name in order to make the object addressable by the process. The address space manager will return to the dynamic linker a unique process-identifier associated with that object. This object identifier has been assumed to be the segment number assigned to the object, however conceptually all an object identifier must do is to allow

the linker (and the user process) to address the object in some fashion.

B. Translator Support

Recall that the linker, using some format or template, must build a linkage table for each object when that object is entered in the process address space. The existence of this template and additionally, the existence of the symbolic name table, imply that the translator used must support dynamic linking by not only translating external references and recognizing entry names, but also by constructing a linkage table template and symbolic name table for the translated object. As will be shown, it is reasonable to assume that a translator can be designed (or modified) with this capability since, in general, all data necessary to construct these two items is either easily computable or available in the translator's symbol table.

1. The Symbolic Name Table

The symbolic name table (Figure 4) contains an entry for each unique external reference and entry name. In addition, the linkage table offset of the corresponding link (i.e., outgoing and incoming links) for each symbolic name table entry should be stored with that entry. We note that this is mandatory for entry names; however, for external references, the addition of the (outgoing) link offset is only convenient since it removes the requirement for the linker to store this information.

It is reasonable to ask where the symbolic name table is located in a process address space. Recall that for the data object <Data>, we have suggested that Data.sym be appended to Data.link. (This implementation allows <Data> to be based at offset zero and to grow dynamically.) We could use this solution for a procedure also; however, since the symbolic name table does not

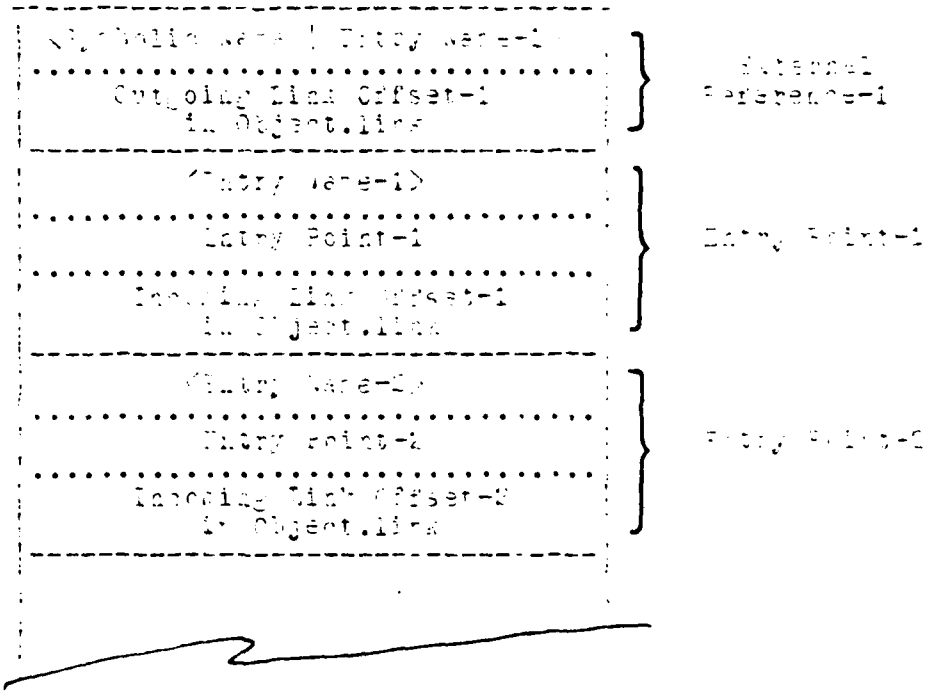


Fig. 4 - Symbol Table

change, a separate copy for each process is not needed. A reasonable solution therefore would be to append the symbolic name table to the end of a (pure) procedure's object code.

2. The Linkage Table Template

The linkage table template (Figure 5) should reflect the exact format of an initialized linkage table with one exception. We have stated that the header of a linkage table contains the virtual address of that object's symbolic name table yet at translation time the segment number of the symbolic name table is unknown. Therefore, we must construct this virtual address when the linkage table of an object is built. If the symbolic name table is a part of the object code, its virtual address can be computed (given we know the offset of the symbolic name table in the object code) when the object's segment number is obtained from the address space manager. If the symbolic name table is a part of the linkage table, we can obtain the linkage table segment number via the linkage address table.

Notice that the header of Figure 5 includes an entry reflecting the size of the linkage table. This represents useful information if an unlinker is implemented (by providing the unlinker with the size of a linkage table to be removed) and may provide useful data when loading a linkage table in a process address space.

By building the symbolic name table first, the construction of the body of a template becomes quite trivial for the translator. We note that there is a one-to-one mapping between entries in the linkage table and the symbolic name table. Therefore the template can be easily constructed by scanning the symbolic name table and initializing a link compatible with each particular symbolic name table entry. As each link is initialized, we can also enter its

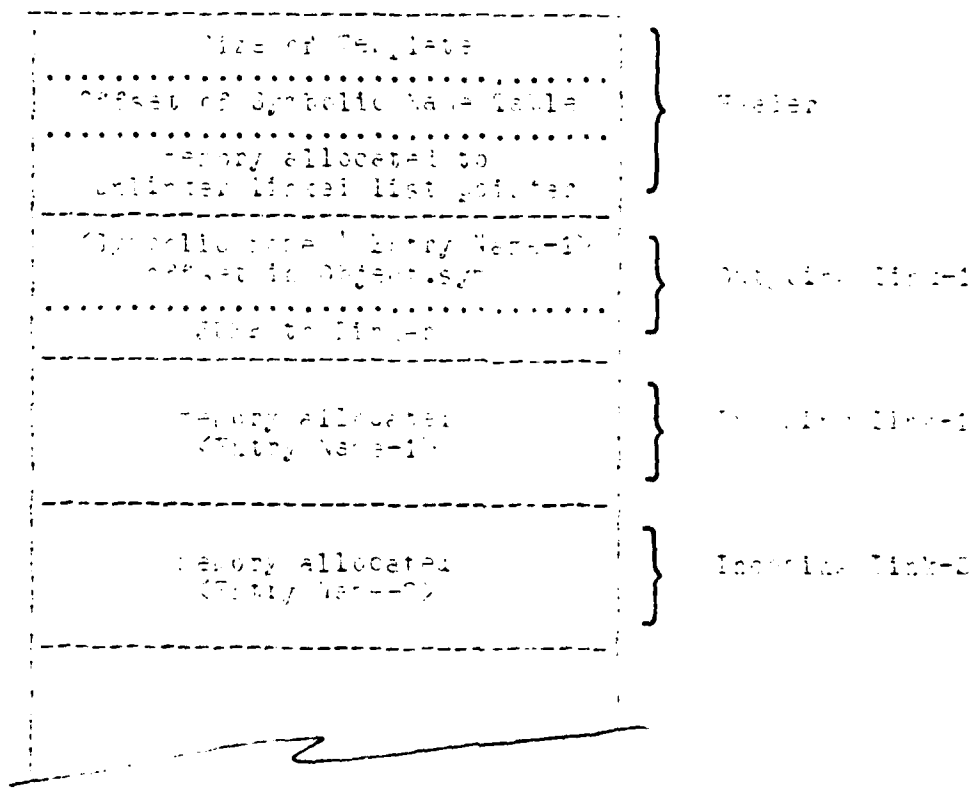


Fig. 5 - Linkage Table Template

link offset into its respective symbolic name table entry.

It is reasonable to ask where in a system the linkage table template should be located. It is suggested that unless demand paging is available, a separate file for the template be created. This will allow the construction of a linkage table without requiring that the template be entered in a process address space. In a demand paging environment, the linkage table template can be a part of the object code and once it has been utilized to construct the linkage table, it will be 'paged out' of main memory since it will not be further referenced.

C. Process Initialization

Much work by Janson [8,9] has been devoted to process initialization including a complete description of how the dynamic linker and the operating system are linked in a multi-domain environment. Our design has been guided by these results. It should be noted that process initialization must include the process reference table and linkage address table. With respect to a user's program, process initialization in a dynamic linking environment differs only in that the linkage table for the user program must be entered in the process address space and the linkage pointer initialized prior to commencing program execution.

V. LINKING WITHOUT TRANSLATOR SUPPORT

Having implied that translator support is necessary to realize dynamic linking, we would like to summarize an implementation in an environment where this condition does not hold; the implementation details have been given elsewhere [10]. It is felt that a linker in this environment should meet certain

fundamental requirements. First, we would like to use the same dynamic linker to link both (translator) supported and unsupported objects thus requiring only one linker and additionally allowing a process to utilize both object types. Also, we do not wish to have to explicitly declare in our source code an external object to be supported or unsupported since, if that object is retranslated with a type change (e.g., unsupported to supported), all procedures referring to that object would have to be retranslated also. (This implies that the linker must be able to differentiate between supported and unsupported objects.)

A. The Interface Module

Recall that we have offered an explicit call to the linker in a procedure's source code as a mechanism for linker invocation such as

CALL <LINK>(Target)

In this example, <LINK> is a module which is statically linked to a process and serves as an interface between the process and the dynamic linker. Conceptually, <LINK> must carry out those functions which, in a (translator) supported system involve the translator.

B. Linking Unsupported Objects

We will now investigate the steps necessary to link the unsupported object <Target> to the unsupported procedure <Caller> (Figure 6).

When <LINK> is called it will enter <Target> in the next free location in Caller.sym and will build an initialized outgoing link for <Target> in Caller.link. (If <Target> already has an entry in Caller.sym, it also has a snapped link and all that is required is to execute the snapped link.) <LINK> will then load <Target>'s parameters into the system according to translator

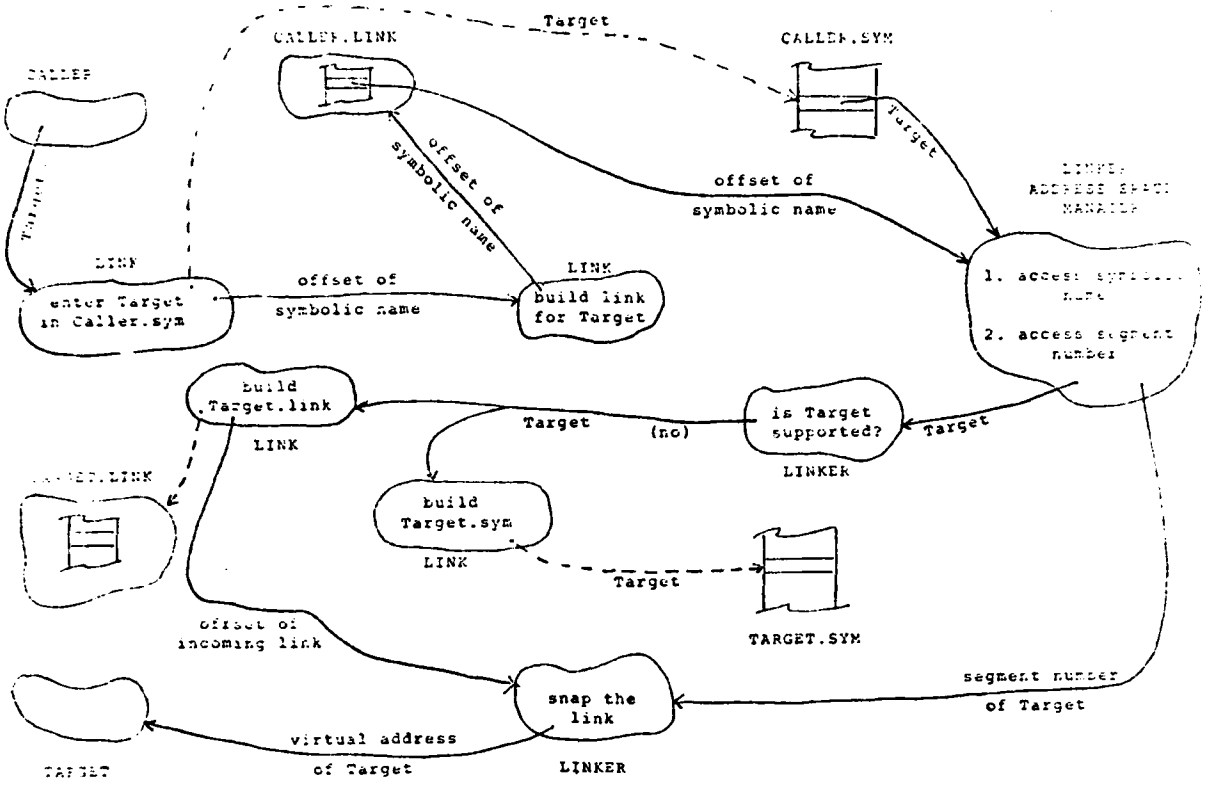


Fig. 6 - Sequence of events for linking unsupported objects.

conventions and execute the outgoing link.

When the linker identifies <Target> as unsupported, it will cause a block of virtual memory to be allocated to serve as Target.link and Target.sym (since <Target> has neither a symbolic name table nor linkage table template). Once Target.link and Target.sym have been initialized, the linker will enter <Target> as the only entry name in Target.sym and complete the link by snapping the incoming link in Target.link (provided <Target> is a procedure).

C. The Interface Linkage Pointer

We cannot expect that an unsupported translator will know that the linkage pointer hardware register is not available for general use and therefore must assume that it cannot be used to point to the linkage table of an executing unsupported procedure. We therefore propose that an interface linkage pointer be established (in software) by <LINK> to duplicate the functions of the hardware linkage pointer.

We should like to observe two important points. First, the incoming link to <Target> must set the interface linkage pointer to point to Target.link vice the hardware linkage pointer. Second, if a process executes both supported and unsupported procedures, the first procedure executed must be supported since this ensures that the hardware linkage pointer is saved before it can be subject to general use by an unsupported procedure.

D. The Disadvantages of Unsupported Dynamic Linking

There are four major disadvantages when linking in an unsupported environment. To begin with, subsequent references to a linked object will be executed much slower than in a supported system since housekeeping functions associated with linking are carried out by software routines vice object

(machine) code. Secondly, we have not proposed a mechanism to pass external procedures to subroutines as actual parameters. A third disadvantage relates to multiple entry points. Since the identification of entry names and their associated entry points requires translator support, an unsupported object can only be accessed at one entry point (viz., its starting location). A fourth disadvantage is that external data will be limited to a based variable similar to those found in PL/I or PL/M since <LINK> can only return to the point of call a pointer to the external data.

VI. HARDWARE IMPLICATIONS

Having discussed the design of a dynamic linker to execute on currently available microprocessors, we would like to examine hardware features which are advantageous in a dynamic linking environment. The hardware features discussed can be classified into two general groups: those which influence the design of the dynamic linker and those which affect the system performance by improving execution speed.

A. Hardware Features Affecting Linking Design

We will discuss four hardware features which affect the design of a dynamic linker. To begin with, the ability to invoke the linker via a hardware fault ensures that a completely general design can be implemented. Without this capability, a link must be snapped when external data is passed as an actual parameter to a subroutine (viz., when the call is executed), implying that a link may be snapped prior to first reference. The second desirable hardware feature involves the capability to reference external objects via indirect addressing. If this feature exists, snapped outgoing

links are simply virtual addresses for indirect addressing instructions. (Multics utilizes indirect addressing and a linkage fault on indirection in the implementation of its dynamic linker [6].)

A third influencing feature involves the size of (hardware) virtual memory. We note if an adequate number of segments exist (assuming each segment is of reasonable size), it may not be necessary to implement an unlinker. (We can always conserve segment numbers by combining smaller objects into one segment prior to execution and referencing each object via an entry point.) The final feature has already been mentioned and is the ability to save and restore the linkage pointer as a part of the microprocessor's CALL and RETURN conventions.

3. Performance Considerations

Along the lines of system performance we will discuss two hardware features: the hardware relocatability of code and hardware segmentation. These are supported by modern microprocessors such as the Zilog Z8000 [7] and the Intel 8096 successor [11].

With respect to hardware relocatability, we note that one must have relocatable procedure segments to implement a dynamic linker since at the start of program execution, there exist no bindings between procedures and process virtual addresses. Therefore, the more efficiently procedures can be relocated (viz., hardware relocation), the better system performance will be.

We have maintained that a dynamic linker can be implemented without hardware segmentation as long as individual objects can be referenced as logical entities (i.e., segments). This implies that many functions intrinsic to a segmented system must be available in a dynamic linking environment. It is only logical, therefore, to conclude that it is advantageous to have hardware

segmentation. In particular, the in-core sharing of objects (e.g., pure procedures) in a multiprogramming environment is extremely difficult to implement without the support of hardware segmentation.

VII. CONCLUSIONS

We have presented a design that illustrates the feasibility of dynamic linking in a microcomputer environment. We have shown that there are only modest demands on the supporting language translators, operating system, and hardware. On the other hand, the design is not in conflict with the more sophisticated, state-of-the-art capabilities that can be expected for microcomputers of the future. Furthermore, the performance cost is quite moderate: the operations for the first reference are essentially those of the traditional linking loader. Subsequently we require about three additional instructions per external call and about two per external data reference. Thus, we conclude that it is possible and practical to include the substantial benefits of dynamic linking in the set of capabilities for present and future microcomputers.

BIBLIOGRAPHY

[1] Presser, L., and White, J. R., "Linkers and Loaders," ACM Computing Surveys, v. 4, p. 149-167, September 1972.

[2] Shaw, A. C., The Logical Design of Operating Systems, Prentice-Hall, 1974.

[3] Madnick, S. E., and Donovan, J. J., Operating Systems, McGraw-Hill, 1974.

[4] Daley, R. C., and Dennis, J. B., "Virtual Memory, Processes, and Sharing in Multics," Communications of the ACM, v. 11 No. 5, May 1968.

[5] O'Connell, J. S., and Richardson, L. D., Distributed, Secure Design for a Multi-Microprocessor Operating System, Master's Thesis, Naval Postgraduate School, June 1979.

[6] Organick, E. I., The Multics System: An Examination of Its Structure, MIT Press, 1972.

[7] Peuto, B. L., "Architecture of a New Microprocessor," Computer, p. 10-21, February 1979.

[8] Janson, P. A., "Dynamic Linking and Environment Initialization in a Multi-Domain Process," Operating System Review, v. 9 No. 5, p. 43-50, November 1975.

[9] Janson, P. A., Removing the Dynamic Linker from the Security Kernel of a Computing Utility, Master's Thesis, Massachusetts Institute of Technology, MIT/MAC TR-132, June 1974.

[10] Blanton, C. B., Dynamic Linking in a Microcomputer Environment, Master's Thesis, Naval Postgraduate School, in preparation.

[11] Markowitz, R., and Pohlman, W. B., "The Evolution Path of the 8086 Microprocessor Architecture for Operating System Environments," Intel Corporation, 1980.

INITIAL DISTRIBUTION LIST

Defense Documentation Center Cameron Station Alexandria, VA 22214	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93940	2
Office of Research Administration Code 012A Naval Postgraduate School Monterey, CA 93940	1
LT Gerald B. Blanton Code 52 Department of Computer Science Naval Postgraduate School Monterey, CA 93940	10
LTCOL Roger R. Schell Code 52Sj Department of Computer Science Naval Postgraduate School Monterey, CA 93940	10
Assistant Professor Bruce MacLennan Code 52Ml Department of Computer Science Naval Postgraduate School Monterey, CA 93940	1
Professor Gordon H. Bradley, Chairman Code 52Bz Department of Computer Science Naval Postgraduate School Monterey, CA 93940	30